GRAPHS

JUNE 15, 2021

Table of Contents

Introduction 1

Degree of Graph: 1

Walks, Circuits, Paths and Cycles in Graph 2

1. Walk 2
2. Circuit 3
3. Path 4
4. Cycle 5

Graph Representation 5

1. Adjacency matrix 5
2. Adjacency list 6

Comparison of adjacency list and matrix: 7

Breadth First Search of Graph 8

Depth First Search of Graph 9

Problems 10

1. Count connected component of graph 10
2. Detect cycle in undirected graph 12
3. Detect cycle in directed graph 14
4. Topological sort (Kahn's algorithm) 16
5. Shortest Path in an Unweighted Undirected Graph 18
6. Shortest path in DAG (Directed Acyclic Graph) 20
7. Minimum spanning Tree (MST) 20

What is Minimum Spanning Tree? 20

Prim's algorithm 21

8. Dijkstra's shortest path algorithm 22
9. Strongly Connected Components 24

Kosaraju's Algorithm 24

10. Bellman Ford Algorithm 27

11.1. Articulation Points (or Cut Vertices) in Graph 29

11.2. Bridges in Graph 33

Problems left with 34

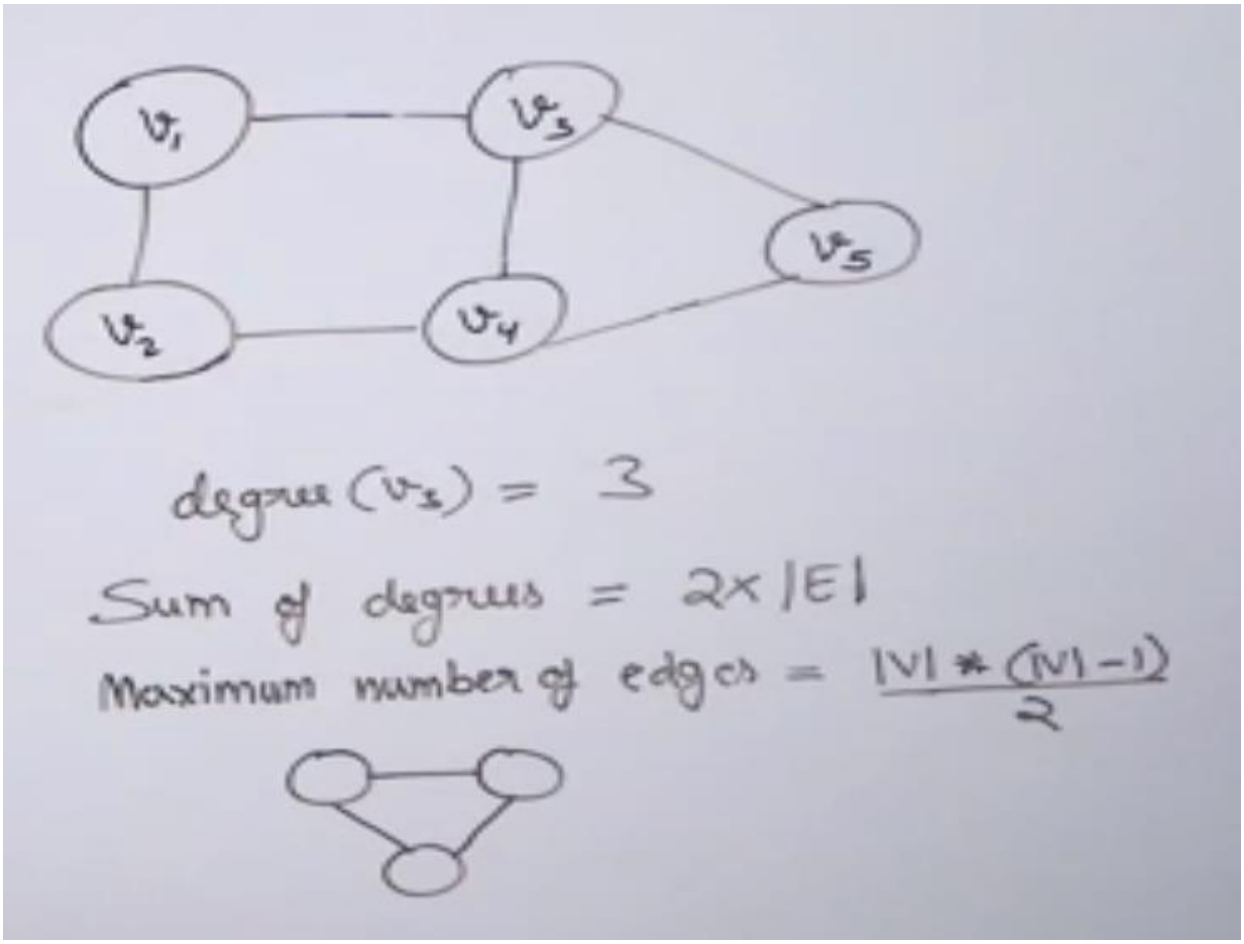
Introduction

A ***Graph*** is a data structure that consists of the following two components:

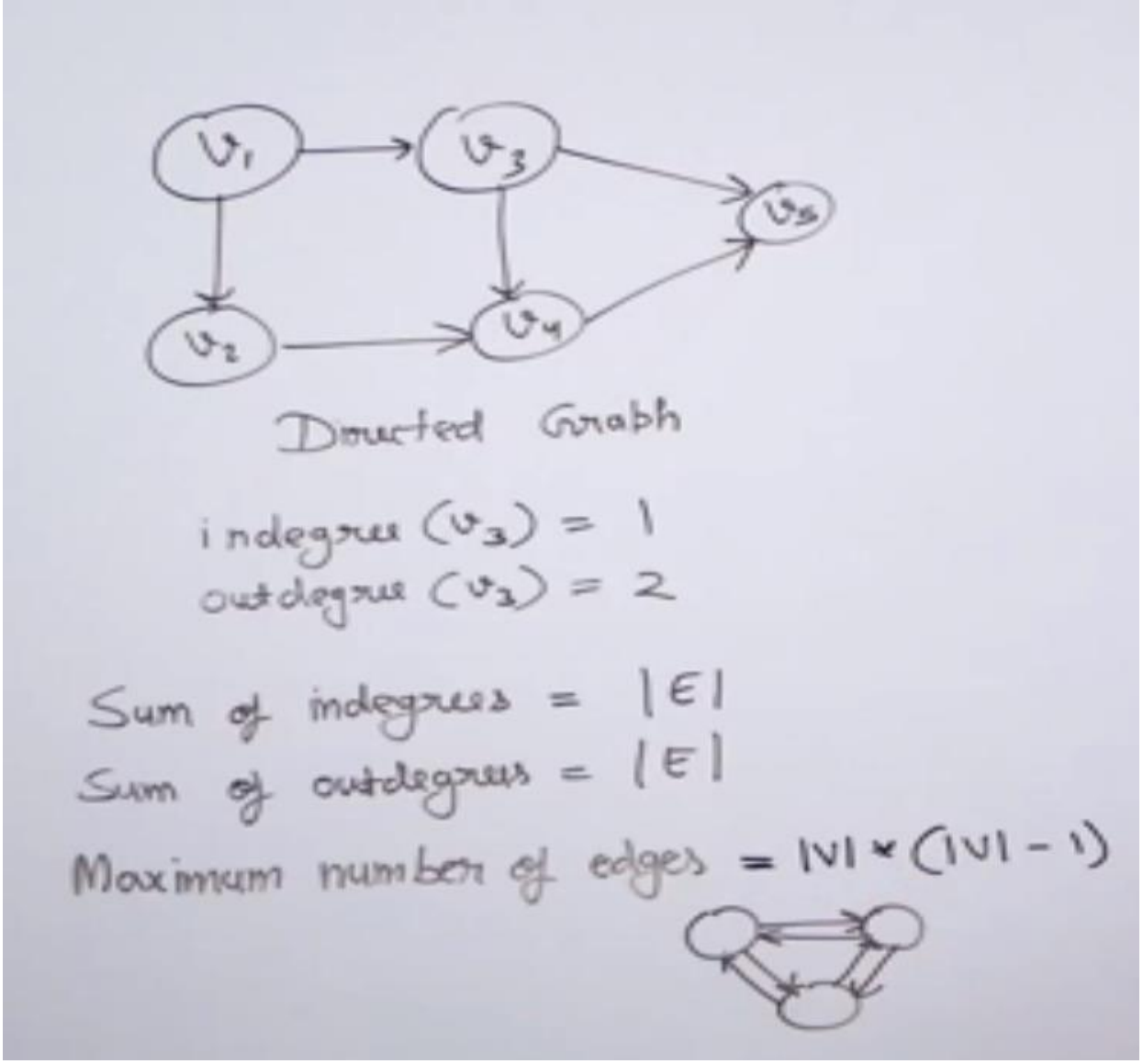
1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Degree of Graph:

1. Degree of undirected graph:



2. Degree of directed graph:



In general, number of edges,

Undirected graph: $E \ll \frac{|V|*(|V|-1)}{2}$

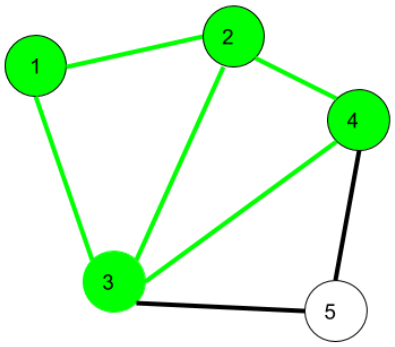
Directed graph: $E \ll |V| * (|V| - 1)$

Walks, Circuits, Paths and Cycles in Graph

1. Walk

A walk is a sequence of vertices and edges of a graph i.e., if we traverse a graph then we get a walk.

- Vertex can be repeated
- Edges can be repeated
- `Walk can repeat anything (edges or vertices).`



Here 1->2->3->4->2->1->3 is a walk

Open walk-A walk is said to be an open walk if the starting and ending vertices are different i.e., the origin vertex and terminal vertex are different.

1->2->3->4->5->3-> is an open walk.

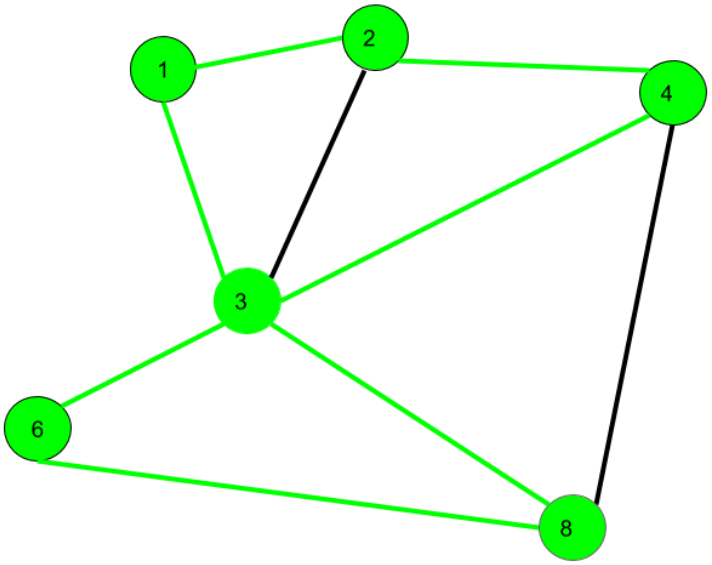
Closed walk-A walk is said to be a closed walk if the starting and ending vertices are identical i.e., if a walk starts and ends at the same vertex, then it is said to be a closed walk.

1->2->3->4->5->3->1-> is a closed walk.

2. Circuit

Traversing a graph such that not an edge is repeated but vertex can be repeated.

- Vertex can be repeated
- Edge not repeated

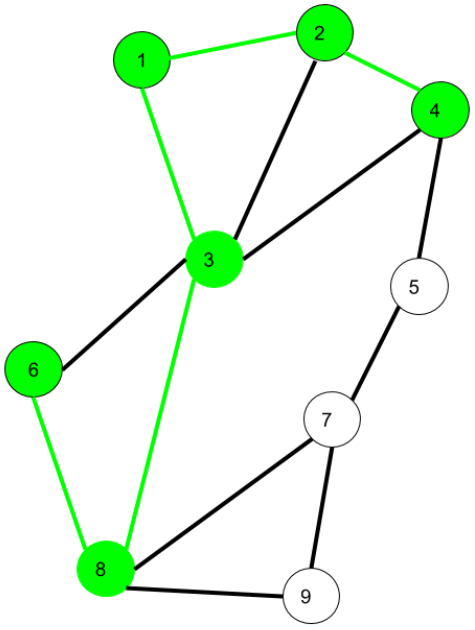


Here 1->2->4->3->6->8->3->1 is a circuit

3. Path

Neither vertices nor edges are repeated i.e., if we traverse a graph such that we do not repeat a vertex and nor we repeat an edge.

- It is also an open walk
- Vertex not repeated
- Edge not repeated

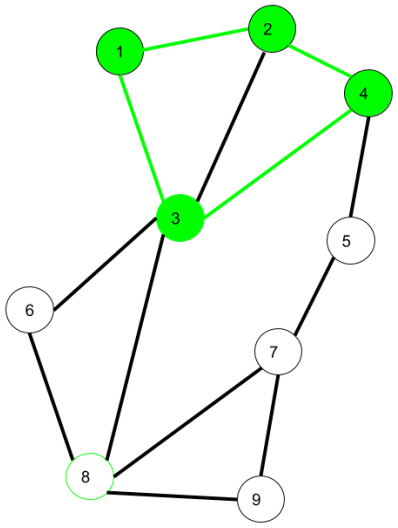


Here 6->8->3->1->2->4 is a Path

4. Cycle

Traversing a graph such that we do not repeat a vertex nor we repeat a edge but the starting and ending vertex must be same i.e. we can repeat starting and ending vertex only then we get a cycle.

- Vertex not repeated
- Edge not repeated



Here 1->2->4->3->1 is a cycle.

Graph Representation

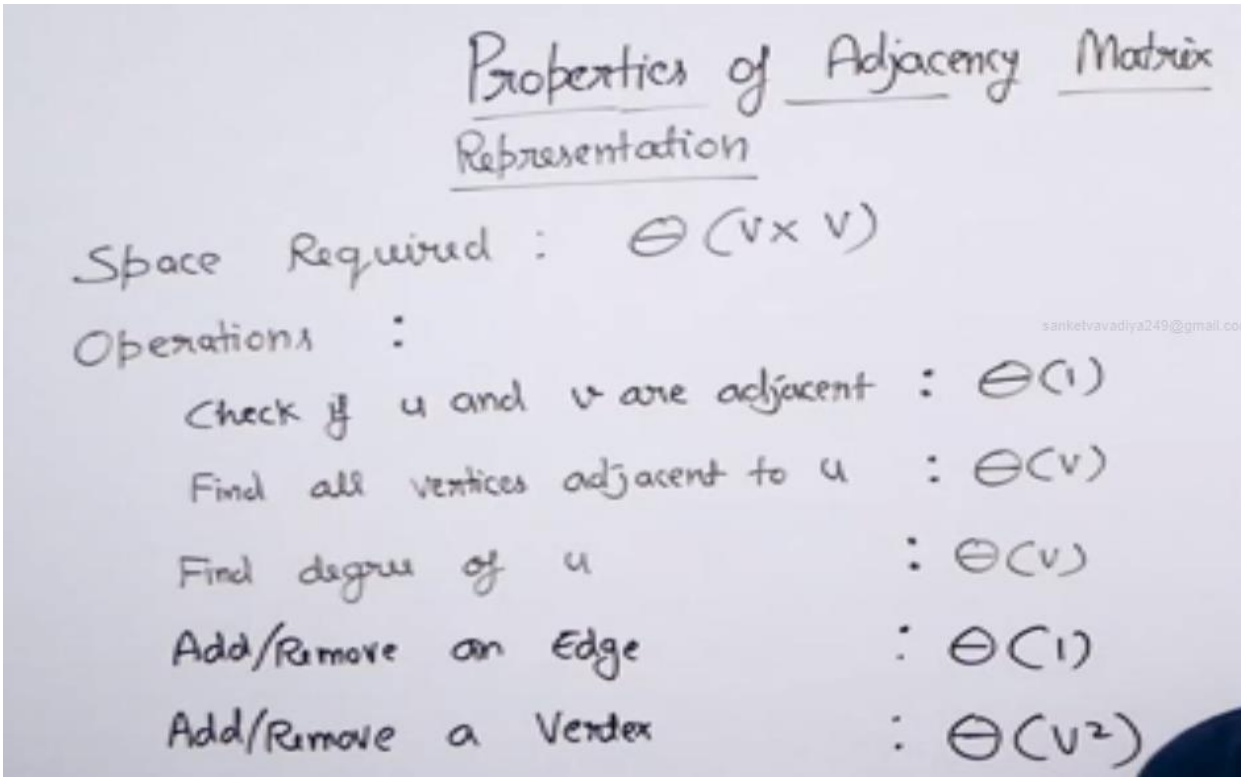
1. Adjacency matrix

The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs.

If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



To add/remove vertex from graph takes $O(n^2)$ time. In both the we first need to create new matrix and then copy all the element of matrix to new matrix.

***Pros**:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

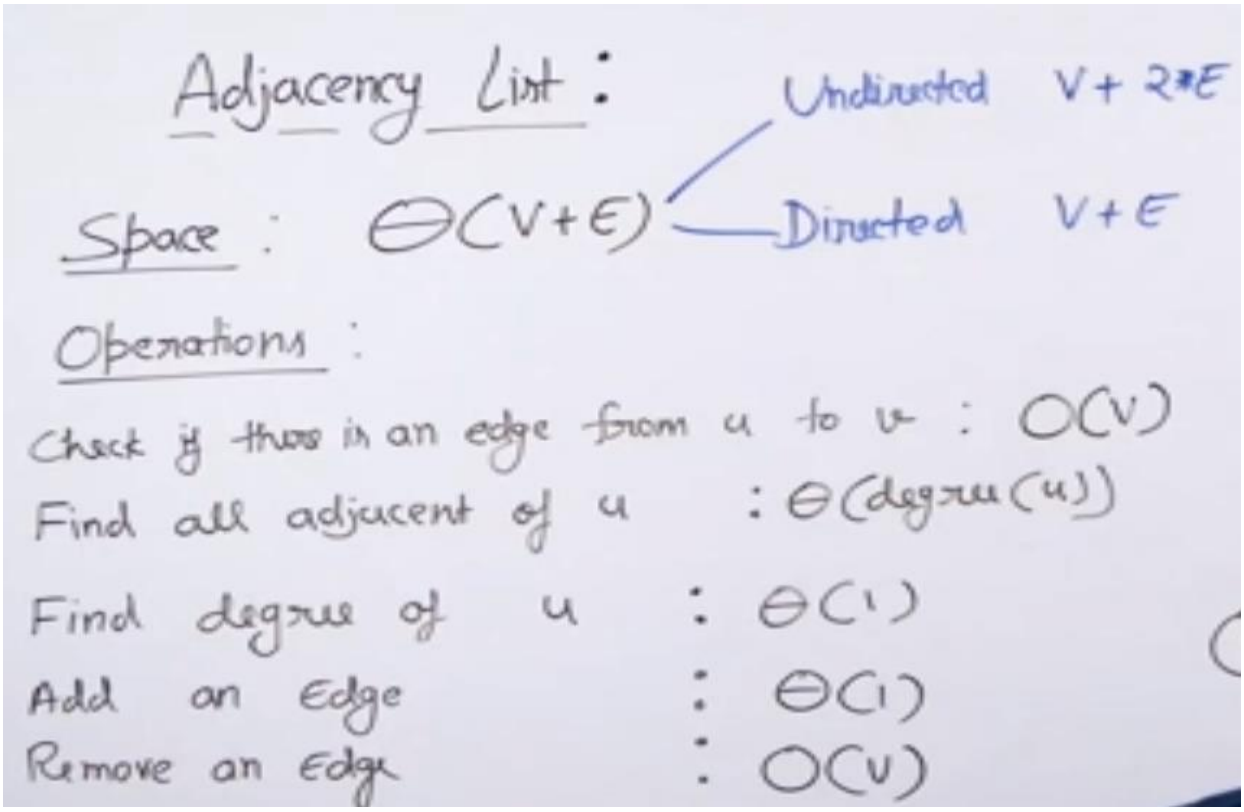
***Cons**:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

2. Adjacency list

Graph can also be implemented using an array of lists.

That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[].

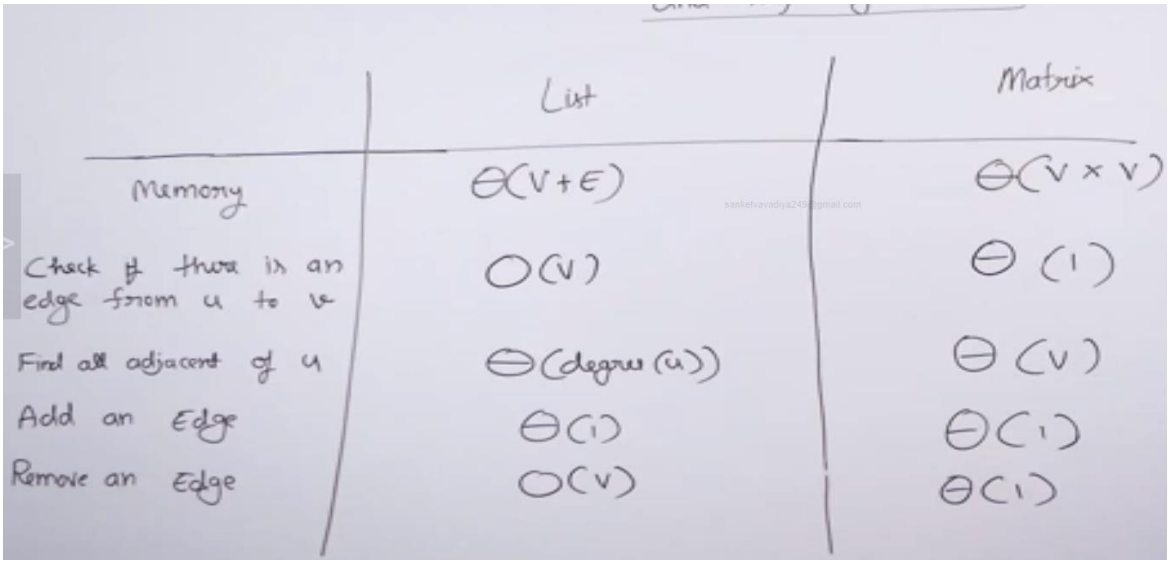
An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



***Pros**:* Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

***Cons**:* Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Comparison of adjacency list and matrix:



Breadth First Search of Graph

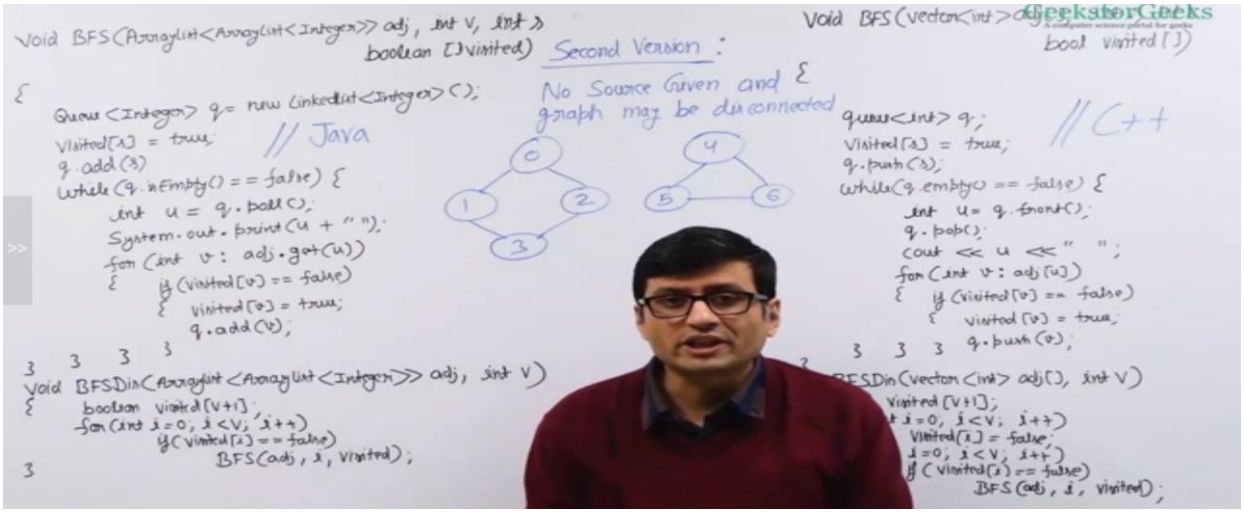
The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.

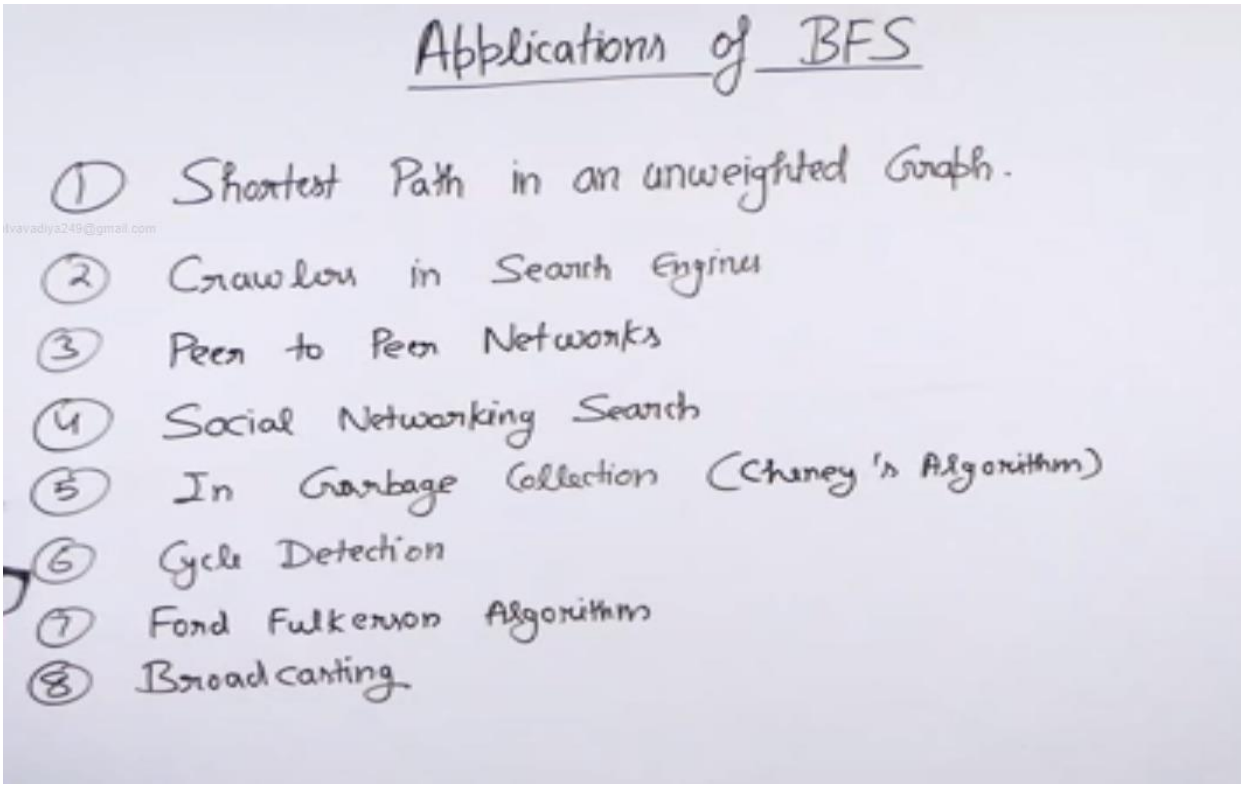
The BFS traversal uses an auxiliary boolean array say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

Complete Algorithm:

1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
3. Until the Queue is non-empty, repeat the below steps:
    - Pop an element from the queue and print the popped element.
    - Traverse all of the vertices adjacent to the vertex poped from the queue.
    - If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

If graph has more than one connected component





Depth First Search of Graph

The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, it in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.

```
class Solution
{
    //Function to return a list containing the DFS traversal of the graph.
    ArrayList<Integer> res;
    boolean [] visited;
    public ArrayList<Integer> dfsOfGraph(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // Code here
        res =  new ArrayList<>();
        visited = new boolean[adj.size()];
        dfs(adj, 0);
        return res;
    }
    void dfs(ArrayList<ArrayList<Integer>> adj, int vertex){
        res.add(vertex);
        visited[vertex] = true;
        ArrayList<Integer> curr = adj.get(vertex);
        for(int i : curr)
            if(!visited[i])
                dfs(adj, i);
    }
}
```

Problems

1. Count connected component of graph

Also, useful to find number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

4. Find the number of islands

Medium Accuracy: 38.66% Submissions: 62137 Points: 4

```
Input:
grid = {{0,1},{1,0},{1,1},{1,0}}
Output:
1
Explanation:
The grid is-
0 1
1 0
1 1
1 0
All lands are connected.
```

Example 2:

```
Input:
grid = {{0,1,1,1,0,0,0},{0,0,1,1,0,1,0}}
Output:
2
Expanation:
The grid is-
0 1 1 1 0 0 0
0 0 1 1 0 1 0
There are two islands one is colored in blue
and other in orange.
```

Using BFS:

```java
class Solution
{
    //Function to find the number of islands.
    class Pair{
        int r, c;
        Pair(int r, int c){
            this.r = r;
            this.c = c;
        }
    }
    public int numIslands(char[][] grid)
    {
        // Code here
        int m = grid.length, n = grid[0].length;
        int count = 0;
        int directions[][] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}, {1, 1}, {-1, 1}, {1, -1}, {-1, -1}};

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(grid[i][j]=='1'){
                    count++;

                    Queue<Pair> q = new LinkedList<>();
                    grid[i][j] = '0';
                    q.add(new Pair(i, j));

                    while(!q.isEmpty()){
                        Pair curr = q.remove();

                        int x = curr.r, y = curr.c;
                        for(int []d : directions){

                            int r = x+d[0], c = y+d[1];

                            if(r>=0 && c>=0 && r<m && c<n && grid[r][c]=='1'){
                                grid[r][c] = '0';
                                q.add(new Pair(r, c));
                            }
                        }
                    }
                }
            }
        }

        return count;
    }

}
```

Using DFS:

```
class Solution
{
    //Function to find the number of islands.
    public int numIslands(char[][] grid)
    {
        // Code here
        int m = grid.length, n = grid[0].length;
        int count = 0;

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(grid[i][j]=='1'){
                    bfs(grid, i, j, m, n);
                    count++;
                }
            }
        }

        return count;
    }

    void bfs(char grid[][], int i, int j, int m, int n){
        if(i>=0 && j>=0 && i<m && j<n && grid[i][j]=='1'){
            grid[i][j] = '0';
            bfs(grid, i+1, j, m, n);
            bfs(grid, i-1, j, m, n);
            bfs(grid, i, j+1, m, n);
            bfs(grid, i, j-1, m, n);
            bfs(grid, i+1, j+1, m, n);
            bfs(grid, i-1, j-1, m, n);
            bfs(grid, i-1, j+1, m, n);
            bfs(grid, i+1, j-1, m, n);
        }
    }
}
```

2. Detect cycle in undirected graph

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

BFS approach:

```java
class Solution
{
    //Function to detect cycle in an undirected graph.
    class Relation{
        int vertex, parent;
        Relation(int vertex, int parent){
            this.vertex = vertex;
            this.parent = parent;
        }
    }

    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // Code here
        boolean []visited = new boolean[V];
        for(int i=0;i<V;i++){
            if(!visited[i]){
                Queue<Relation> q = new LinkedList<>();
                q.add(new Relation(i, i));

                while(!q.isEmpty()){
                    Relation curr_vertex = q.remove();
                    visited[curr_vertex.vertex] = true;

                    ArrayList<Integer> adjecents = adj.get(curr_vertex.vertex);

                    for(int adjecent : adjecents){
                        if(!visited[adjecent])
                            q.add(new Relation(adjecent, curr_vertex.vertex));
                        else if(adjecent!=curr_vertex.parent)
                            return true;
                    }
                }
            }
        }
        // cycle does not exist
        return false;
    }
}
```

DFS approach

```java
class Solution
{
    //Function to detect cycle in an undirected graph.
    boolean cycle;
    boolean visited[];
    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // Code here
        cycle = false;
        visited = new boolean[V];

        for(int i=0;i<V;i++)
            if(!visited[i])
                dfs(adj, i, i);

        return cycle;
    }
    void dfs(ArrayList<ArrayList<Integer>> adj, int vertex, int parent){
        ArrayList<Integer> curr = adj.get(vertex);
        visited[vertex] = true;

        for(int i : curr){
            if(!visited[i])
                dfs(adj, i, vertex);
            else if(parent!=i)
                cycle = true;
        }
    }
}
```

3. Detect cycle in directed graph

Approach use in undirected graph won't work in directed graph.

We need to keep track of ancestors of current vertex. If from the current vertex there is an edge to of its ancestors (parent) then there exist cycle in graph.

DFS approach

Boolean array recursion_stack used to keep track of recursion call stack. Which is used to check parents of current vertex.

```java
class Solution
{
    //Function to detect cycle in a directed graph.
    boolean visited[], recursion_stack[];
    public boolean isCyclic(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // code here
        visited = new boolean[V];
        recursion_stack = new boolean[V];

        for(int i=0;i<V;i++)
            if(!visited[i] && dfs(adj, i))
                return true;

        return false;
    }

    boolean dfs(ArrayList<ArrayList<Integer>> adj, int vertex){
        visited[vertex] = true;
        recursion_stack[vertex] = true;

        ArrayList<Integer> adjecents = adj.get(vertex);
        for(int i : adjecents){
            if(!visited[i] && dfs(adj, i))
                return true;
            else if(recursion_stack[i])
                return true;
        }
        recursion_stack[vertex] = false;
        return false;
    }
}
```

BFS approach

BFS approach is based on Kahn's algorithm. Kahn's algorithm is used for fining topological order of graph.

Algorithm:

1. Store indegree of each vertex in array
2. Enqueue all the vertices having indegree zero.
3. While q is not empty
    a. Remove first element from queue
    b. Mark this popped element as visited
    c. Check for all the neighbours of popped element if indegree is zeros enqueue that neighbour
4. If all vertices are visited there is no cycle in directed graph hence return false. If any of the vertices remain not visited return true as there exists cycle.

```java
class Solution
{
    //Function to detect cycle in a directed graph.
    public boolean isCyclic(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // code here
        int [] indegree = getIndegreeOfVertices(V, adj);
        boolean visited[] = new boolean[V];

        Queue<Integer> q = new LinkedList<>();
        for(int i=0;i<V;i++)
            if(indegree[i]==0)
                q.add(i);

        int visited_vertices = 0;

        while(!q.isEmpty()){
            int popped = q.remove();

            visited[popped] = true;
            visited_vertices++;
            ArrayList<Integer> curr = adj.get(popped);

            for(int i : curr){
                indegree[i]--;
                if(!visited[i] && indegree[i]==0)
                    q.add(i);
            }
        }

        return V!=visited_vertices;
    }
    int[] getIndegreeOfVertices(int V, ArrayList<ArrayList<Integer>> adj){
        int [] indegree = new int[V];
        for(int i=0;i<V;i++){
            ArrayList<Integer> neighbours = adj.get(i);
            for(int n : neighbours)
                indegree[n]++;
        }
        return indegree;
    }
}
```

4. Topological sort (Kahn's algorithm)

BFS approach

Kahn's algorithm is used for fining topological order of graph.

Algorithm:

1. Store indegree of each vertex in array
2. Enqueue all the vertices having indegree zero.
3. While q is not empty
    a. Remove first element from queue
    b. Print popped element
    c. Mark this popped element as visited
    d. Traverse all the neighbours all popped element
        i. Decrement of indegree of current neighbour

ii. If indegree is zero
    1. Add element to queue

```java
class Solution
{
    //Function to return list containing vertices in Topological order.
    static int[] topoSort(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // add your code here
        int [] indegree = getIndegreeOfVertices(V, adj);
        int []res = new int[V];
        boolean visited[] = new boolean[V];
        int last = 0;

        Queue<Integer> q = new LinkedList<>();

        for(int i=0;i<V;i++)
            if(indegree[i]==0)
                q.add(i);

        while(!q.isEmpty()){
            int popped = q.remove();
            visited[popped] = true;
            res[last++] = popped;

            ArrayList<Integer> curr = adj.get(popped);
            for(int i : curr){
                indegree[i]--;
                if(indegree[i]==0)
                    q.add(i);
            }
        }
        return res;
    }

    static int[] getIndegreeOfVertices(int V, ArrayList<ArrayList<Integer>> adj){
        int [] indegree = new int[V];

        for(int v=0;v<V;v++){
            ArrayList<Integer> curr = adj.get(v);
            for(int i : curr)
                indegree[i]++;
        }
        return indegree;
    }
}
```

DFS approach

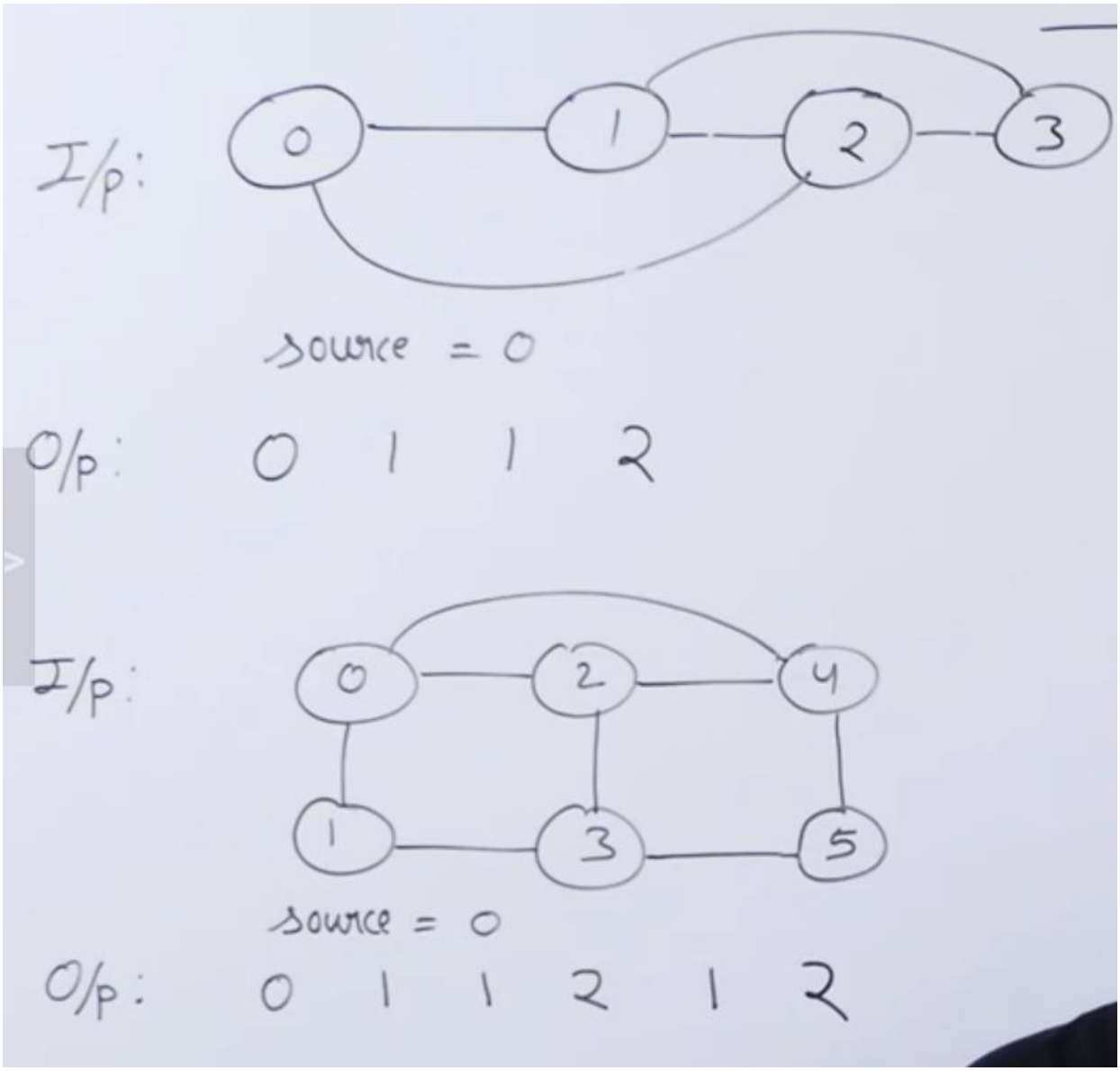
The idea is to do simple DFS. After completion of one DFS (after exiting from recursive call) we need to vertex to stack.

After visiting all the vertices pop one element from stack and print it until all the element of stack popped.

```java
class Solution
{
    //Function to return list containing vertices in Topological order.
    static int[] topoSort(int V, ArrayList<ArrayList<Integer>> adj)
    {
        // add your code here
        Stack<Integer> st = new Stack<>();
        boolean []visited = new boolean[V];

        for(int i=0;i<V;i++)
            if(!visited[i])
                dfs(adj, visited, st, i);

        int res[] = new int[V];
        int last = 0;

        while(!st.isEmpty())
            res[last++] = st.pop();

        return res;
    }

    static void dfs(ArrayList<ArrayList<Integer>> adj, boolean []visited, Stack<Integer> st, int vertex){
        visited[vertex] = true;

        ArrayList<Integer> curr = adj.get(vertex);
        for(int i : curr)
            if(!visited[i])
                dfs(adj, visited, st, i);

        st.push(vertex);
    }
}
```

5. Shortest Path in an Unweighted Undirected Graph

Problem: Given undirected-unweighted graph, Start from the any source vertex. Return an array that shows shortest path from source to that vertex.



Approach:

Initialize resultant array as distance INF for all vertex.

Use BFS traversal, increment distance by one during each next level of graph.

```java
static void BFS(ArrayList<ArrayList<Integer>> adj,int V,int s, int[] dist)
{
    boolean[] visited=new boolean[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    Queue<Integer> q=new LinkedList<>();

    visited[s] = true;
    q.add(s);

    while(q.isEmpty()==false)
    {
        int u = q.poll();

        for(int v:adj.get(u)){
            if(visited[v]==false){
                dist[v]=dist[u]+1;
                visited[v]=true;
                q.add(v);
            }
        }
    }
}
```

6. Shortest path in DAG (Directed Acyclic Graph)

Given a DAG and source vertex, find minimum distance from source vertex to all other vertices.

Here we can use either Dijkstra's algorithms or Bellman ford algorithm to find shortest path. Dijkstra's algorithm takes $O((E+V)*\log V)$ and Bellman ford algorithm takes $O(V^2)$.

But we can reduce the time complexity to $O(V+E)$ using topological sort. Because graph is acyclic we can always get topological order.

Algorithm:

1. Initialize distance[] array of size V to infinite
2. Set distance[source] = 0
3. Find topological order of graph
4. For every vertex u in topological order
    a. For every edge from u to v
        i. If $distance[v] > distance[u] + weight(u,v)$
            1. Set $\text{distance[v]} = \text{distance[u]} + \text{weight(u, v)}$

7. Minimum spanning Tree (MST)

What is Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight

of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

What are the applications of the Minimum Spanning Tree?

Prim's algorithm

Prim's algorithm is a Greedy algorithm.

It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST; the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Algorithm

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices
    a. Pick a vertex u which is not there in mstSet and has minimum key value.
    b. Include u to mstSet.
    c. Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

How does Prim's Algorithm Work?

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So, the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

```java
class Solution
{
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    static int spanningTree(int V, ArrayList<ArrayList<ArrayList<Integer>>> adj)
    {
        // Add your code here
        int []key = new int[V];
        Arrays.fill(key, Integer.MAX_VALUE);
        key[0] = 0;

        boolean [] m_set = new boolean[V];
        int res = 0;
        for(int count = 0;count<V;count++){
            int u = -1;

            // some edges are ignored here
            for(int i=0;i<V;i++)
                if(!m_set[i] && (u==-1 || key[i]<key[u]))
                    u = i;

            m_set[u] = true;
            res = res + key[u];

            ArrayList<ArrayList<Integer>> curr = adj.get(u);

            // some edges are ignored here
            for(ArrayList<Integer> arr : curr){
                int v = arr.get(0), w = arr.get(1);
                if(!m_set[v] && key[v]>w)
                    key[v] = w;
            }
        }
        return res;
    }
}
```

Time complexity analysis

Time Complexity of the algorithm is $O(V\text{^}2)$ when adjacency matrix representation is used. If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to $O(E \log V)$ with the help of binary heap.

8. Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3. While sptSet doesn't include all vertices

    a. Pick a vertex u which is not there in sptSet and has minimum distance value.

    b. Include u to sptSet.

    c. Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

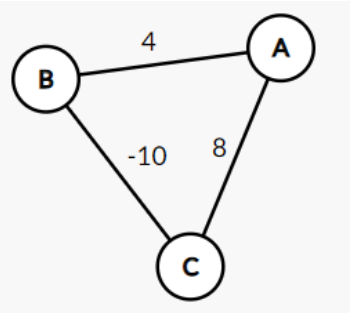
Notes

1. The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it show the shortest path from source to different vertices.
2. The code is for undirected graph, same dijkstra function can be used for directed graphs also.
3. The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).
4. Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap. Please see
5. Dijkstra's algorithm doesn't work for graphs with negative weight cycles, it may give correct results for a graph with negative edges. For graphs with negative weight edges and cycles, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

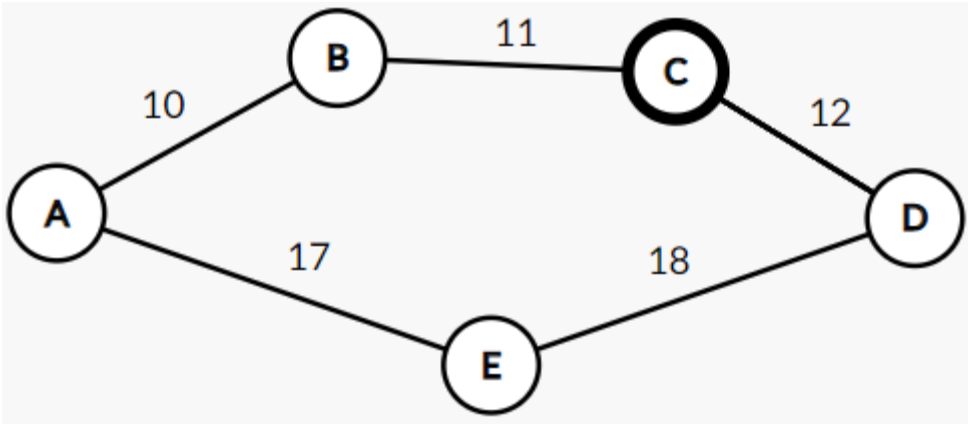
Facts

1. Does not work for negative edge graph

Example:



2. Shortest paths changes if we add weight w to all the edges.



Consider source is A,

Initially, shortest path for D is $A \to B \to C \to D$ with weight w = 33.

After adding 10 weight to each edge pathe shortest path changes to $A \to E \to D$ with weight 55. (Because this change (change in edge weight) is depends on number of edges in path)

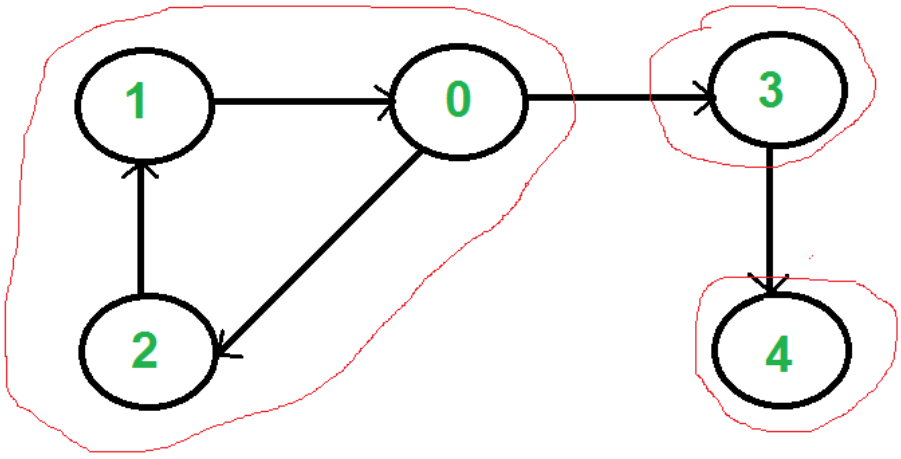
3. Shortest path does not change if we multiply all the edges with some weight.

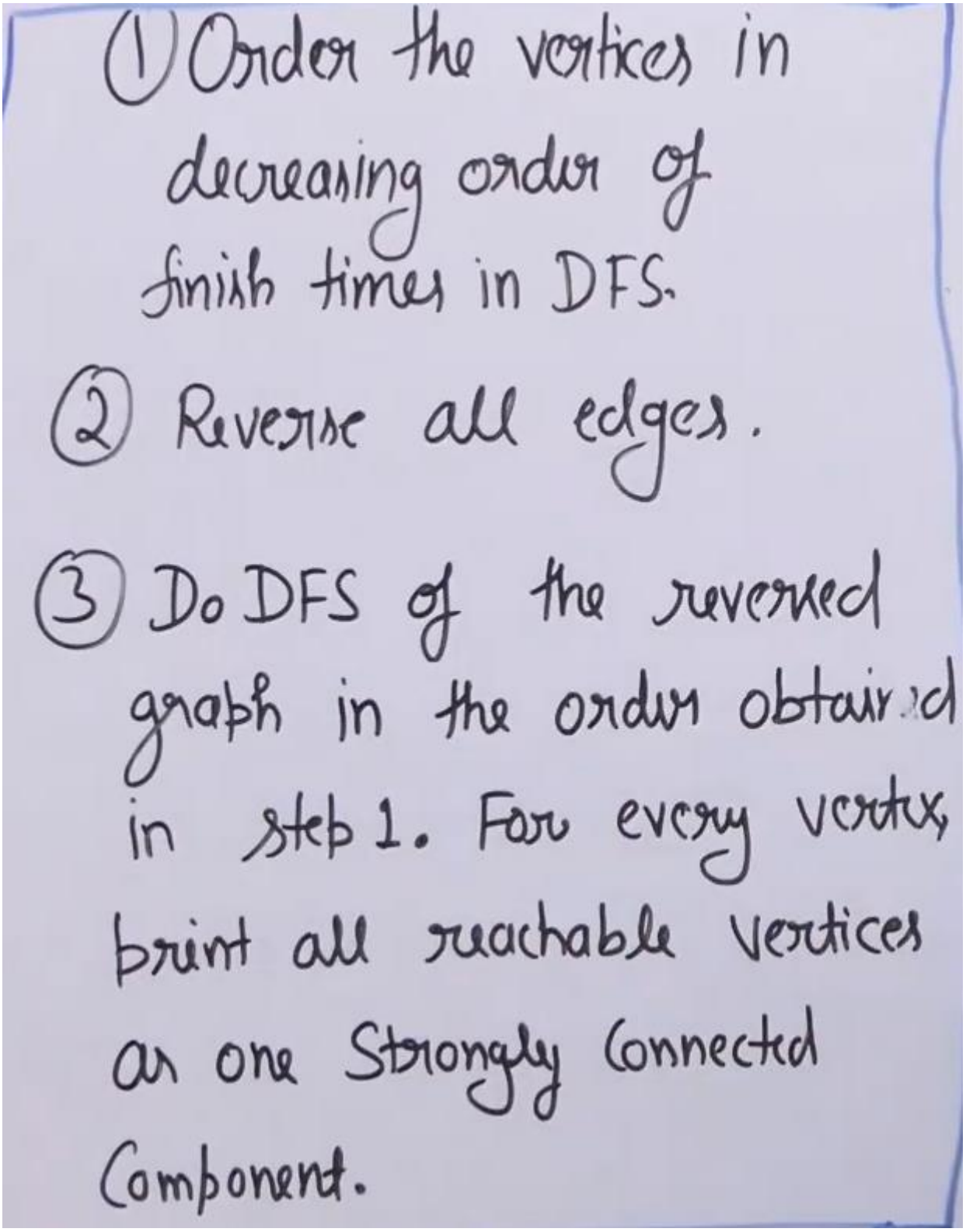
Because common multiplication weight will cancel out.

9. Strongly Connected Components

Kosaraju's Algorithm

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.





We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.

1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.

In other words [during DFS traversal note finish time of all vertices]. In the above graph, if we start DFS from vertex 0, we get vertices in the stack as 1, 2, 4, 3, 0.

2. Reverse all edges of graph to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

```java
public int kosaraju(int V, ArrayList<ArrayList<Integer>> adj)
{
    // Step 1: do dfs and get vertices in decreasing order of their finishing time
    Stack<Integer> st = new Stack<>();
    boolean [] visited = new boolean[V];
    for(int i=0;i<V;i++)
        if(!visited[i])
            dfs(adj, visited, i, st);

    // Step 2: transpose of graph
    adj = reverseEdges(adj, V);

    visited = new boolean[V];
    int component = 0;

    // Step 3: do dfs of transposed graph
    for(int i=0;i<V;i++){
        int popped = st.pop();
        if(!visited[popped]){
            component++;
            dfs(adj, visited, popped);
        }
    }

    return component;
}

// function for Step: 1
void dfs(ArrayList<ArrayList<Integer>> adj, boolean [] visited, int vertex, Stack<Integer> st){
    visited[vertex] = true;

    ArrayList<Integer> curr = adj.get(vertex);
    for(int i : curr)
        if(!visited[i])
            dfs(adj, visited, i, st);
    st.push(vertex);
}

// function for Step: 3
void dfs(ArrayList<ArrayList<Integer>> adj, boolean [] visited, int vertex){
    visited[vertex] = true;
    ArrayList<Integer> curr = adj.get(vertex);

    for(int i : curr)
        if(!visited[i])
            dfs(adj, visited, i);
}
```

```java
 // function for Step: 2
 ArrayList<ArrayList<Integer>> reverseEdges(ArrayList<ArrayList<Integer>> original, int V){
     ArrayList<ArrayList<Integer>> reversed = new ArrayList<>();
     for(int i=0;i<V;i++)
         reversed.add(new ArrayList<Integer>());

     for(int i=0;i<V;i++){
         ArrayList<Integer> curr = original.get(i);

         for(int j : curr)
             reversed.get(j).add(i);
     }
     return reversed;
 }
```

10. Bellman Ford Algorithm

Algorithm: Following are the detailed steps.

- *Input:* Graph and a source vertex *src*.
- *Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, **negative weight cycle is reported.**

1. This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
   Do following for each edge u-v:
   - If dist[v] > dist[u] + weight of edge uv, then update dist[v] as: *dist[v] = dist[u] + weight of edge uv*.
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v.
   - If dist[v] > dist[u] + weight of edge **uv**, then "Graph contains negative weight cycle".

How does this work?

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most **i** edges are calculated.

There can be maximum **|V| - 1** edge in any simple path, that is why the outer loop runs |v| - 1 time.

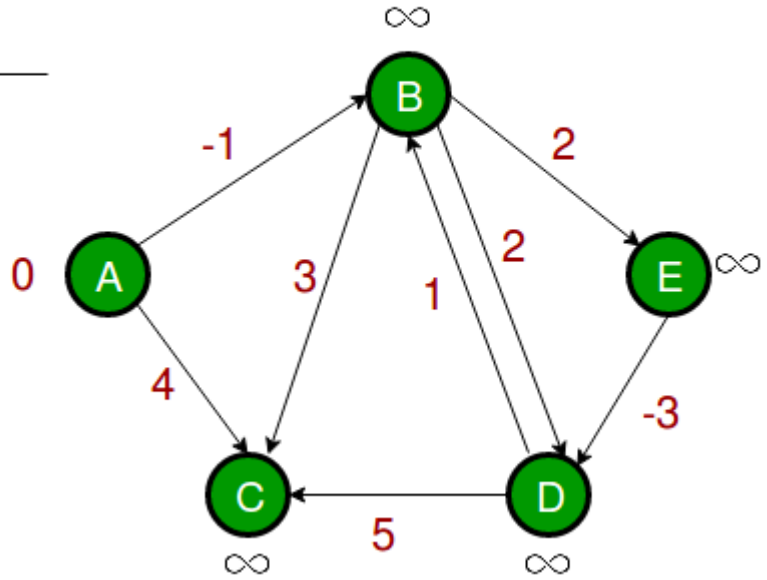
The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edge.

Bellman Ford Algorithm also detects if there is negative weight cycle by doing one more iteration of all the edges (Step 3 in above algorithm is for negative weight cycle).

Example:

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

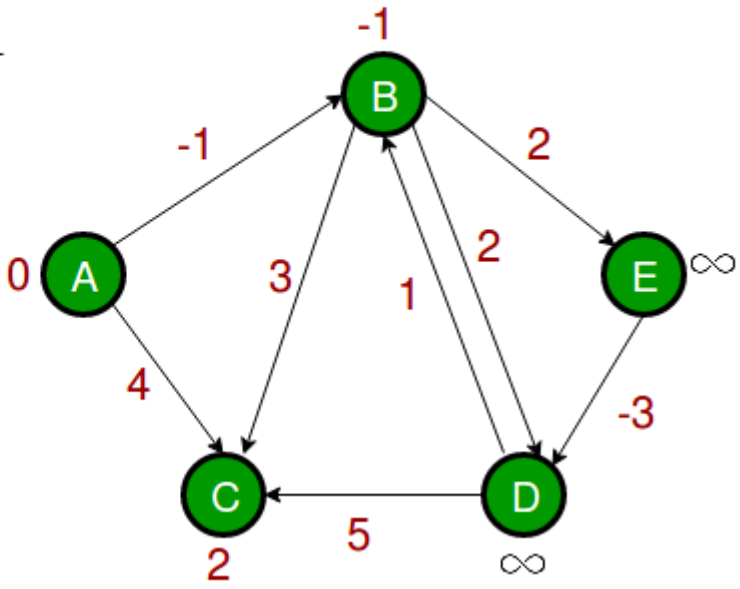
A	B	C	D	E
0	∞	∞	∞	∞



Let all edges are processed in following order: **(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).**

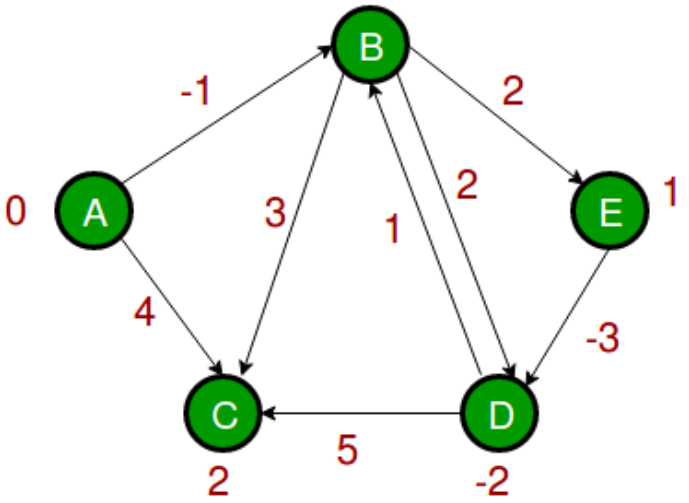
The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed a second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

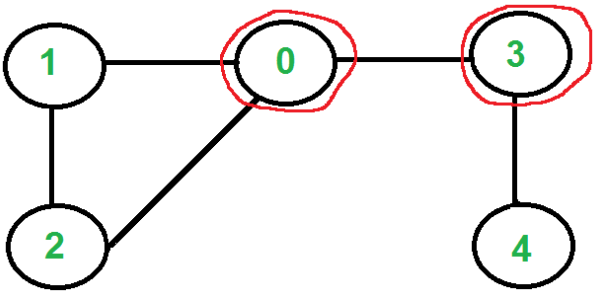
Important Notes:

1. Negative weights are found in various applications of graphs. For example, instead of paying the cost for a path, we may get some advantage if we follow the path.
2. Bellman-Ford works better (than Dijkstra's) for distributed systems. Unlike Dijksra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

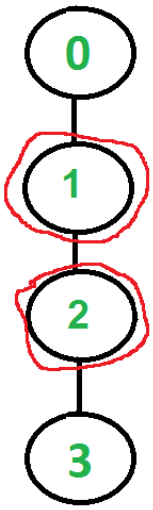
Undirected Unweighted acyclic	BFS	O(V+E)
Directed Acyclic Graph	Topological sort	O(V+E)
Directed/Undirected Cyclic/Acyclic Weighted (Does not work for negative edge weight)	Dijkstra's Algorithm	$O(V^2)$ With Adjacency list and binary heap, it is $O(E \log V)$
Directed/Undirected Cyclic/Acyclic Weighted (Handles negative edge weight)	Bellman ford Algorithm	$O(VE)$ For complete graph where $E = \Theta(V^2)$ it is $O(V^3)$

11.1. Articulation Points (or Cut Vertices) in Graph

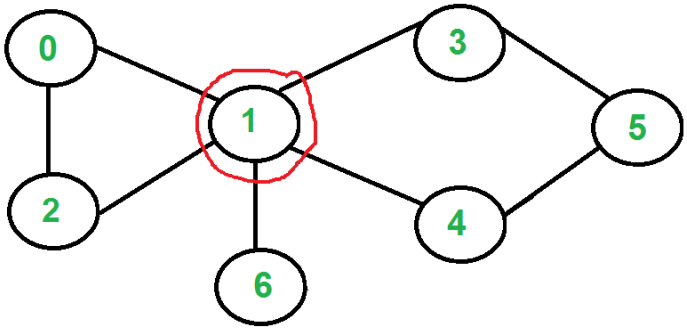
A vertex in an undirected connected graph is an articulation point (or cut vertex) if and only if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.



Articulation points are 0 and 3



Artiulation Points are 1 & 2



Articulation Point is 1

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of a simple approach for the connected graph.

1. For every vertex v, do following:
    - Remove v from graph
    - See if the graph remains connected (We can either use BFS or DFS)
    - Add v back to the graph

The **time complexity** of the above method is O(V*(V+E)) for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Articulation Points (APs). The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is adjacent of u in the graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1. u is the root of DFS tree and it has at least two children.

2. u is not the root of DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.
    - For edge u → v if low[v] ≥ disc[u] then u is AP.

We do DFS traversal of the given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle the second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

```
28      void APUtil(int u, boolean visited[], int disc[],
29                  int low[], int parent[], boolean ap[])
30      {
31
32          int children = 0;
33
34          visited[u] = true;
35
36          disc[u] = low[u] = ++time;
37
38          Iterator<Integer> i = adj[u].iterator();
39          while (i.hasNext())
40          {
41              int v = i.next();
42              if (!visited[v])
43              {
44                  children++;
45                  parent[v] = u;
46                  APUtil(v, visited, disc, low, parent, ap);
47
48
49                  low[u] = Math.min(low[u], low[v]);
50
51
52                  if (parent[u] == NIL && children > 1)
53                      ap[u] = true;
54
55                  if (parent[u] != NIL && low[v] >= disc[u])
56                      ap[u] = true;
57              }
58
59              else if (v != parent[u])
60                  low[u] = Math.min(low[u], disc[v]);
61          }
62      }
```

```
64      void AP()
65      {
66
67          boolean visited[] = new boolean[V];
68          int disc[] = new int[V];
69          int low[] = new int[V];
70          int parent[] = new int[V];
71          boolean ap[] = new boolean[V];
72
73
74          for (int i = 0; i < V; i++)
75          {
76              parent[i] = NIL;
77              visited[i] = false;
78              ap[i] = false;
79          }
80
81          for (int i = 0; i < V; i++)
82              if (visited[i] == false)
83                  APUtil(i, visited, disc, low, parent, ap);
84
85          for (int i = 0; i < V; i++)
86              if (ap[i] == true)
87                  System.out.print(i+" ");
88      }
```

11.2. Bridges in Graph

To find bridge in graph we need to change one condition from Articulation point.

```
28      void bridgeUtil(int u, boolean visited[], int disc[], int low[], int parent[])
29      {
30
31          visited[u] = true;
32
33          disc[u] = low[u] = ++time;
34
35          Iterator<Integer> i = adj[u].iterator();
36          while (i.hasNext())
37          {
38              int v = i.next();
39
40              if (!visited[v])
41              {
42                  parent[v] = u;
43                  bridgeUtil(v, visited, disc, low, parent);
44
45
46                  low[u] = Math.min(low[u], low[v]);
47
48
49                  if (low[v] > disc[u])
50                      System.out.println(u+" "+v);
51              }
52
53
54              else if (v != parent[u])
55                  low[u] = Math.min(low[u], disc[v]);
56          }
57      }
58
59      void bridge()
60      {
61          boolean visited[] = new boolean[V];
62          int disc[] = new int[V];
63          int low[] = new int[V];
64          int parent[] = new int[V];
65
66
67          for (int i = 0; i < V; i++)
68          {
69              parent[i] = NIL;
70              visited[i] = false;
71          }
72
73
74          for (int i = 0; i < V; i++)
75              if (visited[i] == false)
76                  bridgeUtil(i, visited, disc, low, parent);
77      }
```

Problems left with

1. **Minimum Cost Path**
2. **Strongly connected component (Tarjans's Algo)**

```java
class Solution
{
    int time;
    int [] disc, low, parent;
    boolean [] visited, ancesters;
    ArrayList<ArrayList<Integer>> res;
    Stack<Integer> st;

    public ArrayList<ArrayList<Integer>> tarjans(int V, ArrayList<ArrayList<Integer>> adj)
    {
        time = 0;
        disc = new int[V];
        low = new int[V];
        parent = new int[V];
        visited = new boolean[V];
        ancesters = new boolean[V];
        res = new ArrayList<>();
        st = new Stack<>();

        for(int i=0;i<V;i++)
            if(!visited[i]){
                parent[i] = -1;
                dfs(adj, i, V);
            }

        return res;
    }
```

```
    void dfs(ArrayList<ArrayList<Integer>> adj, int u, int V){
        visited[u] = true;
        ancesters[u] = true;
        disc[u] = low[u] = ++time;
        st.push(u);

        ArrayList<Integer> neighbour = adj.get(u);
        for(int v : neighbour){
            if(!visited[v]){
                parent[v] = u;

                dfs(adj, v, V);

                low[u] = Math.min(low[u], low[v]);
            }
            else if(ancesters[v])
                low[u] = Math.min(low[v], low[u]);
        }

        if(disc[u]==low[u])
            addComponent(u);
        ancesters[u] = false;
    }

    void addComponent(int u){
        ArrayList<Integer> one_component = new ArrayList<>();
        res.add(one_component);
        while(!st.isEmpty()){
            int popped = st.pop();
            one_component.add(popped);
            if(popped==u)
                break;
        }
    }

}
```